ELSEVIER
Science of Computer Programming 39 (2001) 3–29
Programming
www.elsevier.nl/locate/scico

Code motion in the presence of critical edges without bidirectional data flow analysis ☆

Oliver Rüthing *

Universität Dortmund, Fachbereich Informatik, Baroper Str. 301, 44221 Dortmund, Germany

Abstract

Bidirectional data flow analysis has become the standard technique for solving bit-vector-based code motion problems in the presence of critical edges. Unfortunately, bidirectional analyses are conceptually and computationally harder than their unidirectional counterparts. In this paper we show that code motion in the presence of critical edges can be achieved without bidirectional data flow analyses. This is demonstrated by means of an adaption of our algorithm for lazy code motion (Knoop et al., Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)'92, ACM SIGPLAN Notices, vol. 27, 7, San Francisco, CA, June 1992, pp. 224–234), which is developed from a fresh, specification-oriented view. As the key element to cope with critical edges homogeneity constraints are introduced in order to avoid anomalies during the code motion process. The "critical" variant of lazy code motion is realized by means of three alternative iteration strategies: (1) a "classical" approach using bidirectional analyses, (2) a unidirectional approach which requires that the control flow is enriched by additional shortcut edges and (3) a hybrid approach which combines unidirectional information flow with the side propagation of information.

Keywords: Program optimization; Partial redundancy elimination; Code motion; Bidirectional data flow analysis; Critical edges; Computational complexity

1. Motivation

In data flow analysis equation systems involving bidirectional dependencies, i.e., dependencies from predecessor nodes as well as from successor nodes, are a well-known source for various kinds of difficulties. First, bidirectional equation systems are conceptually hard to understand. Mainly, this is caused by the lack of a corresponding operational specification like it is given by the meet over all path (MOP) solution of a uni-directional data flow problem. Furthermore, Khedker and Dhamdhere

☆ This article is an extended version of [27].

* Tel.: ++49-231-7555807; fax: ++49-231-7555802.
E-mail address: ruething@ls5.cs.uni-dortmund.de (O. Rüthing).

0167-6423/01/$ - see front matter
PII: S0167-6423(00)00016-2

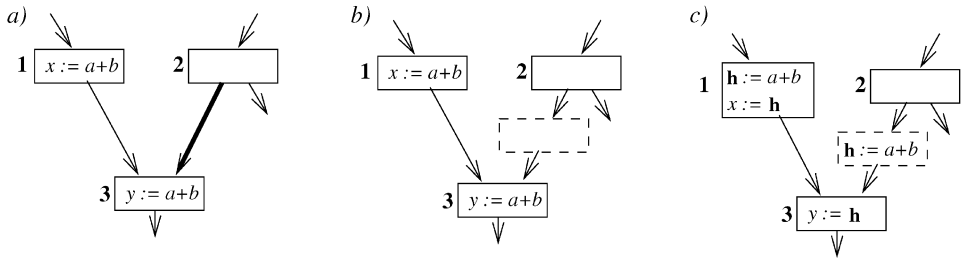

Fig. 1. (a) Critical edge, (b) edge splitting, (c) transformational gain through edge splitting.

recently proved that the computational complexity of solving bidirectional data flow analysis problems is significantly higher than that of solving their unidirectional counterparts [6,15]. This particularly applies to the only practically relevant class of bidirectional analyses, bit-vector-based code motion problems. In fact, all known bidirectional problems are of this kind. Even more specifically, they are more or less variations of Morel's and Renvoise's pioneering algorithm for the elimination of partial redundancies [2–5,9,10,13,14,16–18,21,24,29]. Independently different researchers documented that bidirectionality is only required in programs that have *critical edges* [8,16], i.e. edges in a flow graph that directly lead from branch nodes to join nodes (see Fig. 1(a) for illustration). Ideally, critical edges can be completely eliminated by inserting empty synthetic nodes as depicted in Fig. 1(b). In this example, the additional placement point enables the code motion transformation shown in Fig. 1(c) which eliminates the partial redundant computation on the path through node 1 and 3.[1] However, in practice splitting of critical edges is sometimes not desired since this may introduce additional unconditional jumps or decrease the potential for pipelined execution.[2]

In this paper we investigate a new approach to code motion in the presence of critical edges. This is demonstrated by presenting a "critical" variant of our algorithm for lazy code motion [16]. However, the principal ideas straightforwardly carry over to all related code motion algorithms that employ bidirectional data flow analyses.

Our algorithm is developed from a rigorous, specification oriented view. This particularly allows us to separate between different concerns. While code motion is naturally associated with forward- and backward-oriented propagation of information, the presence of critical edges requires the imposition of additional homogeneity properties which can be expressed in terms of a side propagation of information. Actually, this view allows us to avoid the usage of bidirectional dependencies in our specification. With regard to the variant of lazy code motion the contribution of this paper is threefold:

[1] This is not possible in Fig. 1(a), since hoisting $a+b$ to node 2 introduces a new value on the rightmost path.

[2] Sometimes critical edges are split apart from certain situations that may harm the final code generation.

- On a conceptual level we give a unidirectional specification of the problem. This particularly induces the first MOP characterization of code motion in the presence of critical edges.
- We present a novel hybrid iteration strategy that separates the information flow along critical edges from the information flow along the noncritical ones. While the latter is accomplished by an outer schedule proceeding in standard round-robin discipline the critical information flow is treated exhaustively by an inner schedule.
- Almost as a by-product we obtain the first lifetime optimal algorithm for partial redundancy elimination in the presence of critical edges.

1.1. Related work

As Khedker and Dhamdhere [15] and more recently Masticola et al. [22] noticed, critical edges do not add to the worst-case time complexity of iterative data flow analyses being based on a workset approach. However, this result cannot be generalized to bit-vector analyses where the iteration order has to be organized in a way such that structural properties of the flow graph are exploited in order to take maximum benefit of bit-wise parallel updates through efficient bit-vector operations.

Hecht and Ullman [11] proved an upper bound on the number of round-robin iterations that are necessary for stabilization of monotone, unidirectional bit-vector problems. When proceeding in reverse postorder (or postorder for backward problems) $d+2$ round-robin iterations are sufficient where d is the *depth* of the flow graph, i.e. the maximum number of backedges on an acyclic program path.

Recently, Dhamdhere and Khedker [6,15] generalized this result towards bidirectional problems. However, a major drawback of their setting is that it is pinned to round-robin iterations. Unfortunately, such a schedule does not fit well to situations where information is side-propagated along critical edges. In this light, it is not surprising that their results on the convergence speed of bidirectional bit-vector analyses are quite disappointing. They replace the depth d of a flow graph by its *width* w which is the maximum number of nonconforming edge traversals on an information flow path that does not contain a bypassed subpart.[3] In essence, an information flow path is a sequence of backwards- or forwards-directed edges along which a change of information can be propagated. A forward traversal along a forward edge or a backward traversal along a backward edge are considered conforming with a round-robin schedule proceeding (forwards) in reverse postorder. The other two kind of traversals are nonconforming with respect to this order. Dual notions apply to round-robin iterations proceeding in postorder. Unfortunately, the width is not a structural property of the flow graph, but varies with the problem under consideration, and unlike d which is 0 for acyclic programs it is not even bound in this case. Actually, the notion of width does not match to the intuition associated with the name, as even "slim" programs may have a large width. An intuitive reason for this behaviour is given in Fig. 2(a)

[3] A formal definition is technically extensive. The reader is referred to [6].

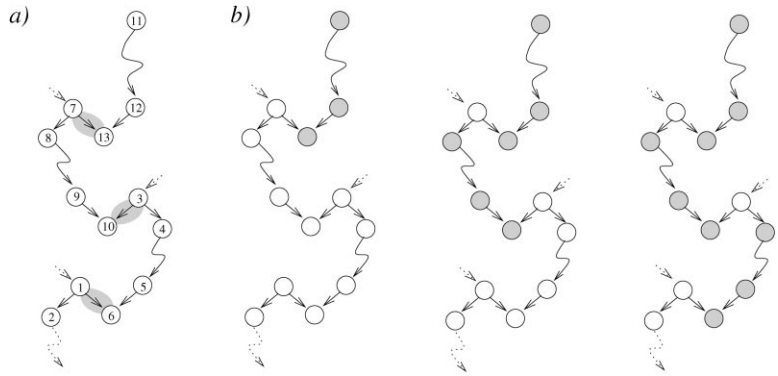

Fig. 2. (a) Acyclic program path responsible for the width of a program. The nodes are numbered in reverse postorder. (b) Slow information propagation in round-robin iterations.

which shows a program fragment with a number of critical edges. Let us assume that information flow in this example follows the equation

$$Info(n) = \sum_{m \in pred(n)} \left(Info(m) + \sum_{n' \in succ(m)} Info(n') \right),$$

which means that the information at node n is set to true if the information at a predecessor or the information of any "sibling" of n is true.

We can easily see that the width of a flow graph with such a fragment directly depends on the number of critical edges, and therefore possibly grows linearly with the "length" of the program. It should be noted that such a program fragment is not totally pathological and thus the linear growth of the width may thus be not unlikely for real-life programs. In fact, considering the reverse postorder of nodes as given in Fig. 2(a) the large width is actually reflected in a poor behavior of a round-robin iteration. Fig. 2(b) shows how the information slowly propagates along the obvious "path" displayed in this example being stopped in each round-robin iteration at a non-conforming (critical) edge. [4]

Dhamdhere and Patil [7] proposed an elimination method for bidirectional problems that is as efficient as in the unidirectional case. However, it is restricted to a quite pathological class of problems, namely weakly bidirectional bit-vector problems and, as usual for elimination methods, it is primarily designed for reducible control flow.

Finally, our hybrid approach shares with the hybrid iteration strategy of Horwitz et al. [12] that it mixes a round-robin schedule with exhaustive subiterations. However, their approach is restricted to the setting of conventional unidirectional bit-vector problems. The subiterations there operate on the strongly connected components of the flow graph

[4] Shaded circles indicate the flow of informations along the "path".

and are used as a pragmatic means to speed up the analyses without actually improving on the worst-case estimation.

2. Preliminaries

We consider programs in terms of *directed flow graphs* $=(N,E,\boldsymbol{s},\boldsymbol{e})$ with node set N, edge set E and unique *start* and *end* nodes $\boldsymbol{s}$ and $\boldsymbol{e}$, respectively. Nodes $n,m,\ldots\in N$ represent (elementary) statements and are assumed to lie on a path from $\boldsymbol{s}$ to $\boldsymbol{e}$. Predecessors and successors of a node $n\in N$ are denoted by $pred(n)$ and $succ(n)$, respectively. and $\boldsymbol{P}[n,m]$ stands for the set of finite paths between node n and m. Finally, for a given finite path p, its length is denoted by ℓ_p and its ith $(1\leqslant i\leqslant\ell_p)$ component by p_i.

2.1. Predicates

2.1.1. Local predicates

As usual our reasoning is based on an arbitrary but fixed expression φ that is the running object for code movement. With each node of the flow graph two local predicates are associated.

$Comp(n)$: φ is computed at n, i.e. φ is part of the right-hand side expression associated with n.

$Transp(n)$: n is *transparent* for φ, i.e. none of φ's variables is modified at n.

2.1.2. Global predicates

Based on these local predicates global program properties are specified. Usually, global predicates are associated with both entries and exits of nodes. In order to keep the presentation simple we assume that every node is split into an entry node and an empty exit node which inherit the set of predecessors and successors from the original node, respectively, and which are assumed to be connected by an edge leading from the entry node to the exit node (Fig. 3). This step allows to restrict our reasoning to entry predicates. It should be noted, however, that this transformation is solely conceptual and does not eliminate any critical edge.

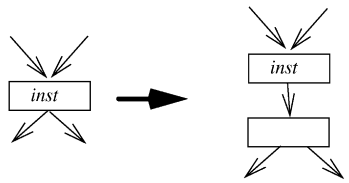

Fig. 3. Splitting of nodes into entry and exit parts.

2.2. Expression motion

In this paper partial redundancy elimination (PRE), or expression motion (EM) as a synonym, stands for program transformations that

(1) insert some instances of initialization statements $\boldsymbol{h}_\varphi := \varphi$ at program points, where $\boldsymbol{h}_\varphi$ is a *temporary variable* that is exclusively assigned to φ and
(2) replaces some original occurrences of φ by a usage of $\boldsymbol{h}_\varphi$.

Hence an expression motion transformation EM is completely characterized by two predicates on nodes in N:

- $Insert_{\mathrm{EM}}$, determining at which program points initializations $\boldsymbol{h}_\varphi := \varphi$ are inserted and
- $Replace_{\mathrm{EM}}$, specifying those program points where an original occurrence of the expression pattern φ is replaced by $\boldsymbol{h}_\varphi$.

2.2.1. Admissibility

In order to guarantee that the semantics of the argument program is preserved, we require that an expression motion transformation must be *admissible*. Intuitively, tuitively, this means that every insertion of a computation is *safe*, i.e. on no program path is the computation of a new value introduced at initialization sites, and that every substitution of an original occurrence of φ by $\boldsymbol{h}_\varphi$ is *correct*, i.e. $\boldsymbol{h}_\varphi$ always represents the same value as φ at use sites. This requires that $\boldsymbol{h}_\varphi$ is properly initialized on every program path leading to some use site in a way such that no modification occurs afterwards. Formally, these properties are captured by the following two predicates. It should be noted that the correctness predicate is parameterized by the transformation EM under investigation.

(1) $$Safe(n) \stackrel{\mathrm{def}}{\Leftrightarrow} \forall p \in \boldsymbol{P}[\boldsymbol{s}, \boldsymbol{e}]\ \forall i \leqslant \ell_p.\ p_i = n\ \Rightarrow \underbrace{\exists j < i.\ Comp(p_j) \wedge \forall j \leqslant k < i.\ Transp(p_k)}_{\text{(i)}}\ \vee \underbrace{\exists j \geqslant i.\ Comp(p_j) \wedge \forall i \leqslant k < j.\ Transp(p_k)}_{\text{(ii)}}.$$

(2) $$Correct_{\mathrm{EM}}(n) \stackrel{\mathrm{def}}{\Leftrightarrow} \forall p \in \boldsymbol{P}[\boldsymbol{s}, n]\ \exists i \leqslant \ell_p.\ Insert_{\mathrm{EM}}(p_i) \wedge \forall i \leqslant j < \ell_p.\ Transp(p_j).$$

Restricting the definition of safety only to the term marked (i) or (ii) induces predicates for *up-safety* and *down-safety*, respectively, which are denoted *UpSafe* and *DnSafe*.

With these definitions the class of admissible expression motion transformations is formally characterized in the following way:

Definition 1. An expression motion transformation EM is *admissible* if and only if for every $n \in N$:

- $Insert_{\mathrm{EM}}(n) \Rightarrow Safe(n)$
- $Replace_{\mathrm{EM}}(n) \Rightarrow Correct_{\mathrm{EM}}(n)$

The set of all admissible expression motion transformations is denoted by $\mathscr{AEM}$.

2.2.2. Computational optimality

The primary goal of expression motion is to minimize the number of computations on every program path. This intent is reflected by the following relation. An expression motion transformation $\mathrm{EM} \in \mathscr{AEM}$ is *computationally the same or better* than an expression motion $\mathrm{EM}' \in \mathscr{AEM}$, in symbols $\mathrm{EM}' \precsim_{exp} \mathrm{EM}$, if and only if

$$\forall p \in \boldsymbol{P}[\boldsymbol{s}, \boldsymbol{e}].\ Comp\#(p, \mathrm{EM}) \leqslant Comp\#(p, \mathrm{EM}'),$$

where $Comp\#(p, \mathrm{EM})$ denotes the number of computations of φ that occur on the path $p \in \boldsymbol{P}[\boldsymbol{s}, \boldsymbol{e}]$ after applying the transformation EM, i.e.

$$Comp\#(p, \mathrm{EM}) \stackrel{\mathtt{def}}{=} |\{i \mid Insert_{\mathrm{EM}}(p_i)\}| + |\{i \mid Comp(p_i) \wedge \neg Replace_{\mathrm{EM}}(p_i)\}|$$

Obviously, $\precsim_{exp}$ defines a preorder on $\mathscr{AEM}$. Based on this preorder we now define:

Definition 2. An admissible expression motion $\mathrm{EM} \in \mathscr{AEM}$ is *computationally optimal* iff $\forall \mathrm{EM}' \in \mathscr{AEM}.\ \mathrm{EM}' \precsim_{exp} \mathrm{EM}$.

Let us denote the set of computationally optimal expression motions by $\mathscr{COEM}$.

2.2.3. Lifetime optimality

The secondary goal of expression motion is to take into account the *lifetime ranges* of the temporaries. Lifetime ranges of temporaries are an important issue in code motion, because unnecessarily large lifetime ranges may increase the *register pressure*. Thus one is primarily interested in those computationally optimal expression motion transformations whose register usage is as economical as possible. A formal definition of lifetime ranges rests on *insertion-replacement paths*, which are paths connecting insertion points with their corresponding replacement sites. More formally, this class is defined by

$$\boldsymbol{IRP}(\mathrm{EM}) \stackrel{\mathtt{def}}{=} \{p \in \boldsymbol{P} \mid Insert_{\mathrm{EM}}(p_1) \wedge Replace_{\mathrm{EM}}(p_{\ell_p}) \wedge \forall 1 < i \leqslant \ell_p.\ \neg Insert_{\mathrm{EM}}(p_i)\}.$$

The set of lifetime ranges

$$LtRg(\mathrm{EM}) \stackrel{\mathtt{def}}{=} \bigcup_{p \in \boldsymbol{IRP}(\mathrm{EM})} \{p_i \mid 1 \leqslant i \leqslant \ell_p\}.$$

induces a *lifetime the same or better* (pre-)order on $\mathscr{AEM}$:

$$\mathrm{EM}' \precsim_{lt} \mathrm{EM} \stackrel{\mathtt{def}}{\Leftrightarrow} LtRg(\mathrm{EM}) \subseteq LtRg(\mathrm{EM}'),$$

which naturally leads to the notion of lifetime optimality.

Definition 3. An expression motion $EM \in \mathcal{COEM}$ is *lifetime optimal* iff $\forall EM' \in \mathcal{COEM}$. $EM' \precsim_{lt} EM$.

3. Expression motion in the absence of critical edges

Before presenting our new approach to PRE in the presence of critical edges we shall first briefly recall the basic steps of lazy expression motion [16] which here serves as a typical representative for the class of Morel/Renvoise-style PRE-algorithms. Furthermore, lazy expression motion is even a particularly powerful representative, as it was the first algorithm for partial redundancy elimination that succeeded in removing partial redundancies as good as possible, while avoiding any unnecessary register pressure. This was achieved by a rigorous redesign of Morel's and Renvoise's algorithm starting from a specification oriented view. The key point in its development was the conceptual separation of the concerns involving computational aspects and the sizes of lifetime ranges. This is reflected in a two-step procedure: lazy expression motion rests on a transformation called busy expression motion.[5] In the following we briefly summarize the details of both transformations.

3.1. Busy expression motion

Busy expression motion (BEM) [10,16,18] places initializations *as early as possible* while replacing all original occurrences of φ. This is achieved by determining the *earliest* program points, of safe program points can be determined by separately computing down-safe and up-safe program points. Both are given through the greatest solutions of two uni-directional data flow analyses, respectively, depicted in Fig. 4.[6]

Then BEM is defined through its insertion and replacement points:

- $Insert_{\mathrm{BEM}}(n) \stackrel{\mathrm{def}}{=} Earliest(n)$.
- $Replace_{\mathrm{BEM}}(n) \stackrel{\mathrm{def}}{=} Comp(n)$.

Despite its surprising simplicity, BEM already reaches *computational optimality*. In fact, as proved in [16,18] we have:

Theorem 4. *BEM is computationally optimal among $\mathcal{AEM}$ when restricted to the universe of programs without critical edges.*

3.2. Lazy expression motion

In addition to BEM, lazy expression motion (LEM) also takes the lifetimes of temporaries into account. This is accomplished by placing initializations *as late as possible*

[5] In [16,18] the transformations are called busy and lazy code motion, respectively. In order to make a clean distinction to assignment motion-based transformations [19,20] we use a more accurate term here.

[6] As common "·","+" and overlining stand for logical conjunction, disjunction and negation, respectively.

$$DnSafe(n) = (n \neq \mathbf{e}) \cdot \Big(Comp(n) + Transp(n) \cdot \prod_{m \in succ(n)} DnSafe(m) \Big)$$

$$UpSafe(n) = (n \neq \mathbf{s}) \cdot Transp(n) \cdot \prod_{m \in pred(n)} Comp(m) + UpSafe(m)$$

$$Safe(n) \stackrel{\mathtt{def}}{=} UpSafe(n) + DnSafe(n)$$

$$Earliest(n) \stackrel{\mathtt{def}}{=} Safe(n) \cdot \Big((n = \mathbf{s}) + \sum_{m \in pred(n)} \overline{Transp(m)} \vee \overline{Safe(m)} \Big)$$

Fig. 4. Computing BEM in the absence of critical edges.

$$Delayed(n) = Earliest(n) + (n \neq \mathbf{s}) \cdot \prod_{m \in pred(n)} Delayed(m) \cdot \overline{Comp(m)}$$

$$Latest(n) \stackrel{\mathtt{def}}{=} Delayed(n) \cdot \Big(Comp(n) + \sum_{m \in succ(n)} \overline{Delayed(m)} \Big)$$

$$Isolated(n) = (n = \mathbf{e}) + \prod_{m \in succ(n)} Earliest(m) + \overline{Comp(m)} \cdot Isolated(m)$$

Fig. 5. Computing LEM in the absence of critical edges.

but *as early as necessary*, where the latter requirement means "necessary in order to reach computational optimality". Technically, this is achieved by determining the *latest* program points, where a BEM-initialization might be *delayed* to. In addition, there is no need to insert a temporary at all, if this temporary would only be used at the same program point immediately afterwards. Such *isolated* program points are identified by means of an additional analysis. Fig. 5 summarizes the additional analyses of LEM. LEM is then defined by

- $Insert_{\mathrm{LEM}}(n) \stackrel{\mathtt{def}}{=} Latest(n)$.
- $Replace_{\mathrm{LEM}}(n) \stackrel{\mathtt{def}}{=} Comp(n) \cdot \overline{Latest(n) \cdot Isolated(n)}$.

As proved in [16,18] we have:[7]

Theorem 5. *LEM is computationally optimal as well as* lifetime optimal *among the transformations in $\mathcal{COEM}$ when restricted to the universe of programs without critical edges.*

[7] In [28] we show that this optimality result is only adequate for flat universes of expressions. If both composite expressions and their subexpressions are moved, then the notion of lifetime optimality changes and a significantly more sophisticated technique has to be applied. Nevertheless, LEM still provides a basic ingredient of this approach.

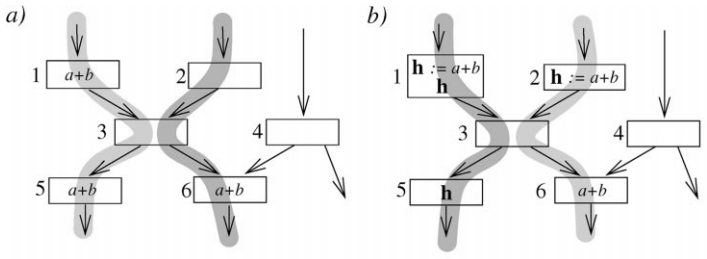

Fig. 6. Incomparable admissible expression motion transformations.

4. Expression motion in the presence of critical edges

In this section our new approach to PRE in the presence of critical edges is elaborated in full details. First we shall investigate the principal differences to the setting presented in Section 3.2. As opposed to flow graphs without critical edges there are usually no computationally optimal representatives. In fact, Fig. 6 shows two admissible, but computationally incomparable transformations that cannot be improved any further. The first one is simply given by the identity transformation of the program in Fig. 6(a), the result of the second one is displayed in Fig. 6(b). Each of the resulting programs has exactly one computation on the path that is emphasized in the dark shade of grey, while having two computations on the path being emphasized in the light shade of grey, respectively. Thus there is no computationally optimal expression motion transformation with respect to the original program in Fig. 6(a).

This problem can be overcome by restricting the range of program transformations to those that are *profitable*, which means those introducing initializations only when they are actually needed on every program path originating at the initialization site. This particularly ensures that such a transformation is computationally the same or better than the identity transformation. Formally, this is captured by the following predicate characterizing profitable insertion points wrt an underlying expression motion transformation EM:

$$\begin{aligned} \mathit{Profitable}_{\mathrm{EM}}(n) \stackrel{\mathrm{def}}{\Leftrightarrow} \forall p \in \boldsymbol{P}[n,\boldsymbol{e}]\ \exists i \leqslant \ell_p.\ \ & \mathit{Replace}_{\mathrm{EM}}(p_i)\ \wedge \\ & \forall 1 < j \leqslant i.\ \neg \mathit{Insert}_{\mathrm{EM}}(p_j). \end{aligned}$$

With this definition the class of profitable expression motion transformations is formally characterized in the following way:

Definition 6. An expression motion transformation $\mathrm{EM} \in \mathscr{AEM}$ is *profitable* iff $\forall n \in N.\ \mathit{Insert}_{\mathrm{EM}}(n) \Rightarrow \mathit{Profitable}_{\mathrm{EM}}(n)$.

We shall denote the set of profitable, admissible expression motion transformations by $\mathscr{PAEM}$. It should be noted that the transformation in Fig. 6(b) is not profitable, as

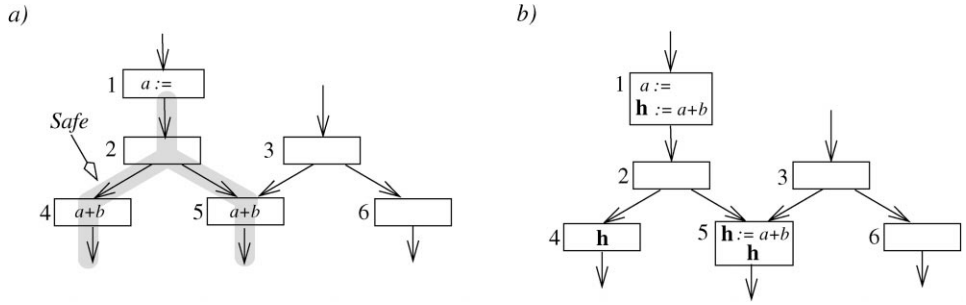

Fig. 7. (a) Range of safe program points, (b) Program degradation through a naive adaption of busy expression motion.

the insertion at node 2 is not used on the path leading through nodes 3 and 6, while the identity transformation in Fig. 6(a) obviously is profitable.

4.1. Busy expression motion

In this section we develop a counterpart to BEM in the presence of critical edges. After briefly sketching the difficulties that prohibit a straightforward adaption of the noncritical solution, a correct approach is systematically developed from a specification that incorporates the special role of critical edges.

Unfortunately, BEM as presented in Section 3.1 cannot straightforwardly be applied to flow graphs with critical edges. This is because such a naive adaption may include nonprofitable transformations. This is illustrated by means of Fig. 7. Based on the range of safe program points as depicted in Fig. 7(a) one would obtain earliest initialization points at nodes 1 and 5 leading to the transformation displayed in Fig. 7(b). This, however, introduces an additional computation on the path leading through nodes 1 and 5, while no path at all is strictly improved.

4.1.1. Homogeneous propagation of down-safety

The key for a useful critical variant of BEM is to impose an additional homogeneity constraint on safety ensuring that safety information either propagates to all or to none of its predecessors. This ensures that earliest program points become a proper upper borderline of the region of safe program points. In fact, in the absence of critical edges safety has the following homogeneity property:

$$\forall n \in N.\ Safe(n) \Rightarrow \begin{cases} \forall m \in pred(n).\ Safe(m) \ \vee, \\ \forall m \in pred(n).\ \neg Safe(m). \end{cases}$$

In the presence of critical edges, however, the homogeneity may be violated. For instance, in Fig. 7(a) node 5 as well as node 2 are safe, while node 3 is not. Therefore, we consider the following notion of *homogeneous down-safety*:

Definition 7. A predicate *HDnSafe* on the nodes of N is a *homogeneous down-safety predicate* iff for any $n \in N$

(1) *HDnSafe* is *conformable* with down-safety

$$HDnSafe(n) \Rightarrow (n \neq \boldsymbol{e}) \cdot \left(Comp(n) + Transp(n) \cdot \prod_{m \in succ(n)} HDnSafe(m) \right).$$

(2) *HDnSafe* is *homogeneous*:

$$HDnSafe(n) \Rightarrow \prod_{m \in pred(n)} HSafe(m) + \prod_{m \in pred(n)} \overline{HSafe(m)},$$

where

$$HSafe(n) \stackrel{\text{def}}{=} HDnSafe(n) + UpSafe(n).$$

Obviously, homogeneous down-safety predicates are closed under "union".[8] Thus there exists a uniquely determined largest homogeneous down-safety predicate $DnSafe_{\mathtt{Hom}}$, which gives rise to a homogeneous version of safety, too:

$$\forall\, n \in N.\ Safe_{\mathtt{Hom}}(n) \stackrel{\text{def}}{=} DnSafe_{\mathtt{Hom}}(n) + UpSafe(n).$$

As in the "noncritical" setting $Comp(n)$ implies $DnSafe_{\mathtt{Hom}}(n)$.

It should be noted that the definition is developed from a pure specification oriented reasoning and can be seen as a first rigorous characterization of down-safety in the presence of critical edges: down safety is described by a backward directed data flow problem which is restricted by additional homogeneity constraints. This is in contrast to other algorithms, where bidirectional equation systems are postulated in an ad hoc fashion without any separation of their functional components.

Earliest program points are defined as in the noncritical case with the only difference of using the homogeneous variant of down-safety in place of the usual one.

$$Earliest_{\mathtt{Hom}}(n) \stackrel{\text{def}}{=} Safe_{\mathtt{Hom}}(n) \cdot \left((n = \boldsymbol{s}) + \sum_{m \in pred(n)} \overline{Transp(m)} + \overline{Safe_{\mathtt{Hom}}(m)} \right).$$

The earliest program points serve as insertion points of BEM for flow graphs with critical edges (CBEM) which is defined by

- $Insert_{\mathrm{CBEM}}(n) \stackrel{\text{def}}{=} Earliest_{\mathtt{Hom}}(n)$.
- $Replace_{\mathrm{CBEM}}(n) \stackrel{\text{def}}{=} Comp(n)$.

Then we have:

Theorem 8. *CBEM is computationally optimal among* $\mathscr{PAEM}$.

A detailed proof is contained in the appendix.

[8] This means the predicate defined by the pointwise conjunction of the predicate values.

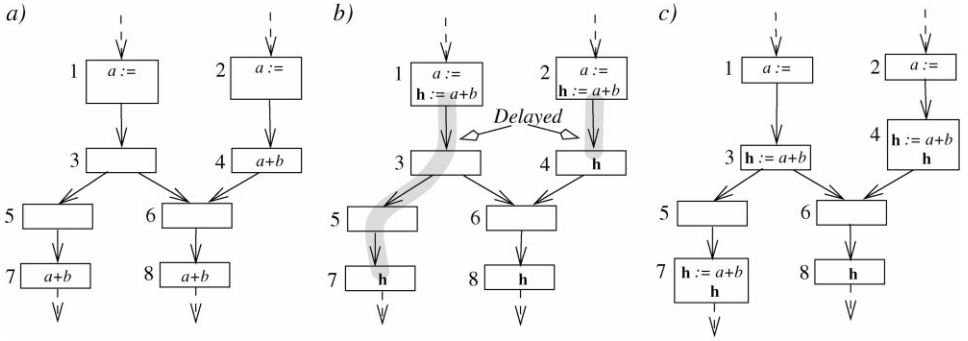

Fig. 8. Program degradation caused by a naive adaption of LEM: (a) the argument program, (b) result of CBEM and the range of delayable program points (c) the result of naive LEM.

4.2. Lazy expression motion

Similar to the situation in Section 4.1 also the relevant analyses of LEM as defined in Section 3.2 cannot naively be adapted to flow graphs with critical edges, even if CBEM is assumed as the basis for the delay process. This phenomenon is illustrated by means of Fig. 8(a). Part (b) of the figure shows the result of CBEM. A naive adaption of the delayability predicate from LEM would then determine delayable program points as emphasized in the figure. Thus initializations at the latest program points would result in the program depicted in Fig. 8(c). Note, however, that this transformation increases the number of computations of $a + b$ on the path $(1, 3, 5, 8, \ldots)$, while no path would be strictly improved.

Like safety in BEM, delayability in LEM is also burdened with homogeneity defects when critical edges are present. In fact, for flow graphs without critical edges we have

$$Delayed(n) \Rightarrow \begin{cases} \forall\, m \in succ(n).\ Delayed(m) \ \vee \\ \forall\, m \in succ(n).\ \neg Delayed(m). \end{cases}$$

This property may now be violated. For instance, in Fig. 8(b) both the exit of node 3 and the entry of node 5 meet the delayability property, whereas the entry of node 6 does not. Hence one has to force homogeneity explicitly in order to yield an appropriate critical variant of lazy expression motion. Therefore, let us consider the following notion of *homogeneous delayability*.

Definition 9. A predicate HDelayed on N is a *homogeneous* delayability predicate iff for any $n \in N$

(1) $HDelayed(n)$ is conformable with delayability:

$$HDelayed(n) \Rightarrow Earliest_{\mathtt{Hom}}(n) + \left((n \neq \boldsymbol{s}) \cdot \prod_{m \in pred(n)} HDelayed(m) \cdot \overline{Comp(m)} \right).$$

(2) $HDelayed(n)$ is homogeneous:

$$Delayed(n) \Rightarrow \prod_{m \in succ(n)} HDelayed(m) + \prod_{m \in succ(n)} \overline{HDelayed(m)}.$$

Like homogeneous down-safety predicates also homogeneous delayability predicates are closed under "union". Thus there exists a unique largest homogeneous delayability predicate $Delayed_{\mathtt{Hom}}$. This gives rise to a new version of latestness characterizing the insertion points of lazy expression motion for flow graphs with critical edges (CLEM).

$$Latest_{\mathtt{Hom}}(n) \stackrel{\mathtt{def}}{\Leftrightarrow} Delayed_{\mathtt{Hom}}(n) \cdot \left(Comp(n) + \sum_{m \in succ(n)} \overline{Delayed_{\mathtt{Hom}}(m)} \right).$$

Now LEM for flow graphs with critical edges (CLEM) is defined:

- $Insert_{\mathrm{CLEM}}(n) \stackrel{\mathtt{def}}{=} Latest_{\mathtt{Hom}}(n)$.
- $Replace_{\mathrm{CLEM}}(n) \stackrel{\mathtt{def}}{=} Comp(n) \cdot \overline{Latest(n) \cdot Isolated(n)}$.

Then we have:

Theorem 10. *CLEM is computationally optimal among $\mathscr{PAEM}$ and lifetime optimal among the computationally optimal programs.*

The proof of this theorem is included in the appendix.

5. Computing CBEM and CLEM

This section is devoted to the issue how the specifying solutions of CBEM and CLEM can be realized by appropriate data flow analyses. We will discuss three alternative approaches:

(1) a "classical" approach via bidirectional analyses,
(2) a new nonstandard approach that transforms the problem into one with purely unidirectional equations, and
(3) a hybrid approach that separates forwards and backwards flow from side propagation of information.

5.1. The bidirectional approach

In this Section we present "classical bidirectional" solutions for CBEM and CLEM. In contrast to other bidirectional expression motion algorithms algorithm postulated in the past [1–3,5,6,8,9,13,14,23,25,30] bidirectionality here is not introduced in an adhoc fashion, but rather has a clear functional role in terms of forcing homogeneity.

$$UpSafe(n) = (n \neq \mathbf{s}) \cdot Transp(n) \cdot \prod_{m \in pred(n)} Comp(m) + UpSafe(m)$$

$$DnSafe_{\mathtt{Hom}}(n) = (n \neq \mathbf{e}) \cdot \Big(Comp(n) + Transp(n) \cdot \prod_{m \in succ(n)} DnSafe_{\mathtt{Hom}}(m)\Big) \prod_{m \in pred(succ(n))} Safe_{\mathtt{Hom}}(m)$$

$$Safe_{\mathtt{Hom}}(n) = UpSafe(n) + DnSafe_{\mathtt{Hom}}(n)$$

$$Earliest_{\mathtt{Hom}}(n) \stackrel{\mathtt{def}}{=} Safe_{\mathtt{Hom}}(n) \cdot \Big((n = \mathbf{s}) + \sum_{m \in pred(n)} \overline{Transp(m)} + \overline{Safe_{\mathtt{Hom}}(m)}\Big)$$

Fig. 9. Computing CBEM: the bidirectional variant.

5.1.1. CBEM

The specification of Definition 7 can be transfered into a bidirectional equation system for down-system as displayed in Fig. 9. For both *UpSafe* and $DnSafe_{\mathtt{Hom}}$ the largest fixed point of the equation system is computed. Note that $Safe_{\mathtt{Hom}}$ is not truly involved in the fixed-point computation but rather serves as an abbreviation. The correspondence between the equation system and the specification is easy to establish. Essentially, it reduces the property formulated in the following lemma, which is easy to prove:

Lemma 11. *Let HDnSafe be a predicate on the nodes in N with*

$$\forall n \in N.\ HDnSafe(n) \Rightarrow (n \neq \boldsymbol{e}) \cdot \left(Comp(n) + Transp(n) \cdot \prod_{m \in succ(n)} HDnSafe(m)\right) \tag{1}$$

and $HSafe(n) \stackrel{\mathtt{def}}{=} HDnSafe(n) + UpSafe(n)$. *Then the following two properties* (2a) *and* (2b) *are equivalent*:

$$\forall n \in N.\ HDnSafe(n) \Rightarrow \prod_{m \in pred(n)} HSafe(m) + \prod_{m \in pred(n)} \overline{HSafe(m)}, \tag{2a}$$

$$\forall n \in N.\ HDnSafe(n) \Rightarrow \prod_{m \in pred(succ(n))} HSafe(m). \tag{2b}$$

At this point it is worth noting that the bidirectional solution shares the problems sketched in Fig. 2 when subjected to a round-robin iteration strategy. In fact, violation of homogeneous down-safety follows exactly the same definition pattern as *Info* does in this example.[9] Hence slow propagation of down-safety would be also apparent in CBEM.

[9] Actually, here nondown safety is propagated in a dual fashion.

$$Delayed_{\mathtt{Hom}}(n) = Earliest_{\mathtt{Hom}}(n) + \big((n \neq \mathbf{s}) \cdot \prod_{m \in pred(n)} Delayed_{\mathtt{Hom}}(m) \cdot \overline{Comp(m)}\,\big) \prod_{m \in succ(pred(n))} Delayed_{\mathtt{Hom}}(m)$$

$$Latest_{\mathtt{Hom}}(n) \stackrel{\mathtt{def}}{=} Delayed_{\mathtt{Hom}}(n) \cdot \big(Comp(n) + \sum_{m \in succ(n)} \overline{Delayed_{\mathtt{Hom}}(m)}\,\big)$$

$$Isolated(n) = (n = \mathbf{e}) + \prod_{m \in succ(n)} Earliest_{\mathtt{Hom}}(m) + \overline{Comp(m)} \cdot Isolated(m)$$

Fig. 10. Computing CLEM: the bidirectional variant.

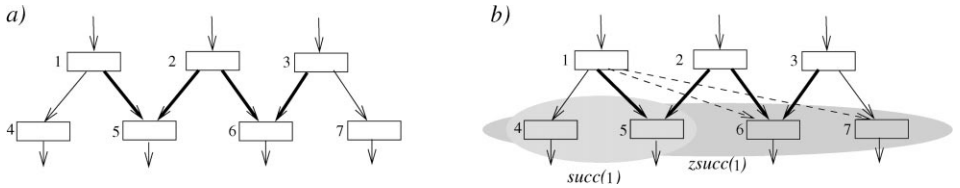

Fig. 11. (a) Program fragment with a nest of critical edges (b) Zig-zag successors and virtual shortcut edges of node 1.

5.1.2. CLEM

Also the specification of Definition 9 can be directly transfered into a bidirectional equation system for delayability which is depicted in Fig. 9. The correspondence between the specification and the equation system is completely dual to one in CBEM (cf. Lemma 11).

5.2. The unidirectional approach

In this section we present how CBEM and CLEM can be transfered in a way that they can be regarded as unidirectional problems (Fig. 10).

5.2.1. CBEM

It is important to note that specification of homogeneous down-safety does not require "true" forward propagation of down-safety information. This is because according to the node splitting assumption of Section 2.1 a join node is always followed by a single successor that does not have any other predecessors (see Fig. 3 for illustration). Rather the forward flow can be viewed as a "side propagation" of down-safety information along zig-zag paths. For a technical description let us define the set of *zig-zag successors* $zsucc(n)$ of a node $n \in N$ as the smallest set of nodes satisfying (see Fig. 11 for illustration):

(1) $succ(n) \subseteq zsucc(n)$.
(2) $\forall m \in zsucc(n).\ succ(pred(m)) \subseteq zsucc(n)$.

$$UpSafe(n) = (n \neq \mathbf{s}) \cdot Transp(m) \cdot \prod_{m \in pred(n)} Comp(m) + UpSafe(m)$$

$$DnSafe_{\mathtt{Hom}}(n) = n \neq \mathbf{e} \cdot \big(Comp(n) + Transp(n) \cdot \prod_{m \in zsucc_{\mathtt{US}}(n)} DnSafe_{\mathtt{Hom}}(m)\big)$$

$$Safe_{\mathtt{Hom}}(n) \stackrel{\mathtt{def}}{=} UpSafe(n) + DnSafe_{\mathtt{Hom}}(n)$$

$$Earliest_{\mathtt{Hom}}(n) \stackrel{\mathtt{def}}{=} Safe_{\mathtt{Hom}}(n) \cdot \big((n = \mathbf{s}) + \sum_{m \in pred(n)} \overline{Transp(m)} + \overline{Safe_{\mathtt{Hom}}(m)}\big)$$

Fig. 12. Computing CBEM: the unidirectional variant.

In essence, in our application we are faced with a zig-zag propagation of information (here nondown-safety) which is further stopped at nodes where up-safety can be established. Hence we introduce a parameterized notion of $zsucc(n)$ which, for $M \subseteq N$, is defined by

(1) $succ(n) \subseteq zsucc_M(n)$.
(2) $\forall m \in zsucc_M(n).succ(pred(m) \backslash M) \subseteq zsucc_M(n)$.

With $\mathtt{US} \stackrel{\mathtt{def}}{=} \{n \in N \mid UpSafe(n)\}$ the equation for down-safety can now be rewritten as shown in Fig. 12 below. Formally, the equivalence of this formulation with the one of Fig. 9 is based on the following lemma whose proof is an easy exercise.

Lemma 12. *Let HDnSafe a predicate with*

$$\forall n \in N.\ HDnSafe(n) \Rightarrow (n \neq \boldsymbol{e}) \cdot \left(Comp(n) + Transp(n) \cdot \prod_{m \in succ(n)} HDnSafe(m)\right) \tag{3}$$

and $HSafe(n) \stackrel{\mathtt{def}}{=} HDnSafe(n) + UpSafe(n)$. *Then the following two properties* (4a) *and* (4b) *are equivalent*:

$$\forall n \in N.\ HDnSafe(n) \Rightarrow \prod_{m \in pred(succ(n))} HSafe(m), \tag{4a}$$

$$\forall n \in N.\ HDnSafe(n) \Rightarrow \prod_{m \in zsucc_{\mathtt{US}}(n)} HDnSafe(m). \tag{4b}$$

Note that this equation system is a completely unidirectional one, however, one that operates on a flow graph being enriched by shortcut edges drawn between nodes and their zig-zag successors (see Fig. 11(b)). An important contribution of the unidirectional approach is that it provides the first "meet over all paths (MOP)" characterization of safety in the presence of critical edges. Actually, only the notion of paths has to be extended towards paths across shortcut edges: a sequence of nodes $(n_1, \ldots, n_k)$ is a

$$Delayed_{\mathtt{Hom}}(n) = Earliest_{\mathtt{Hom}}(n) + \big((n \neq \mathbf{s}) \cdot \prod_{m \in zpred(n)} Delayed_{\mathtt{Hom}}(m) \cdot \overline{Comp(m)}\big)$$

$$Latest_{\mathtt{Hom}}(n) \stackrel{\mathtt{def}}{=} Delayed_{\mathtt{Hom}}(n) \cdot \big(Comp(n) + \sum_{m \in succ(n)} \overline{Delayed_{\mathtt{Hom}}(m)}\big)$$

$$Isolated(n) = (n = \mathbf{e}) + \prod_{m \in succ(n)} Earliest_{\mathtt{Hom}}(m) + \overline{Comp(m)} \cdot Isolated(m)$$

Fig. 13. Computing CLEM: the unidirectional variant.

finite US*-zig-zag path* iff $n_{i+1} \in zsucc_{\mathtt{US}}(n_i)$ for $1 \leqslant i < k$. Denoting the set of finite zig-zag paths between n and m by $\boldsymbol{ZP}_{\mathtt{US}}[n,m]$ the MOP-characterization of down-safety reads as

$$DnSafe_{\mathtt{Hom}}(n) \stackrel{\mathtt{def}}{\Leftrightarrow} \forall p \in \boldsymbol{ZP}_{\mathtt{US}}[n,\boldsymbol{e}]\ \exists i \leqslant \ell_p.\ Comp(p_i) \wedge \forall 1 \leqslant j < i.\ Transp(p_j).$$

5.2.2. CLEM

In analogy to Section 5.2.1 the unidirectional version of delayability is based on a zig-zag variant of predecessors defined by

(1) $pred(n) \subseteq zpred(n)$,
(2) $\forall m \in zpred(n).\ pred(succ(m)) \subseteq zpred(n)$.

However, as opposed to the context of safety where a parameterized notion of zig-zag successors was introduced in order to take into account up-safety information, here we get along with the unparameterized notion. The unidirectional formulation of CLEM is summarized in Fig. 13. Coincidence of the unidirectional and bidirectional variant can be easily established along the lines of Lemma 12.

5.3. The hybrid approach

Although we already provided unidirectional solutions for CBEM and CLEM, in the worst case the analyses would require the insertion of a quadratic number of shortcut edges. In fact, for a zig-zag chain of k critical edges as shown in Fig. 11 the number of shortcut edges is of order k^2. Even though long zig-zag chains of critical edges can be expected to be rare in practice, we will show that information propagation can be organized as efficiently without such blow-up in the number of edges.

To this end we present a hybrid approach that combines unidirectional information flow with explicit side flow of information, while sharing the main advantages of the bidirectional and unidirectional approach:

- like the bidirectional approach it operates on the original flow graph without adding virtual shortcut edges and

- like the unidirectional approach it is mainly driven by the efficient round-robin strategies proceeding along post- or reverse postorder of the nodes.

5.3.1. CBEM

The hybrid variant of CBEM is based on the bidirectional equation system of Fig. 9 which already makes the side flow component explicit. In the following we will give a description of the iteration strategy that better fits to this equation system than the usual round robin strategy proceeding in postorder traversals of the flow graph. The overall schedule of the approach is as follows:

Preprocess. Collapsing of nodes according to the side flow of information.

Outer schedule. Process the collapsed nodes in postorder until stabilization is reached performing an inner schedule.

Inner schedule. (1) For each node within a collapsed component perform information propagation along its outgoing noncritical edges.

(2) Perform exhaustive information propagation along the outgoing critical edges within the collapsed node under investigation.

In the following we will go into the details of this process.

The preprocess. The collapsing step groups together nodes of N according to the following equivalence relation:

$$n \equiv m \overset{\text{def}}{\Leftrightarrow} zsucc(n) = zsucc(m).$$

It should be noted that G can be decomposed into its equivalence class easily by tracing zig-zag paths of critical edges originating at an unprocessed node. For instance, starting with node 1 in Fig. 11(a) we obtain the equivalence class $\{1,2,3\}$ by following the critical edges. Clearly, this process can be managed in order $\mathcal{O}(e)$, where e denotes the number of edges in E. All the nodes of an equivalence class are collapsed into a single node that inherits all incoming and outgoing edges of its members (see Fig. 14 for illustration).

The outer schedule. The flow graph G' that results from the collapsing preprocess is used in order to determine the round-robin schedule which drives information backwards. It should be noted that the depth of G' may differ from the depth of the original flow graph G in both directions: the depth may increase or decrease by collapsing. This is illustrated in Fig. 15. While in Part (a) the depth decreases, since the indicated path

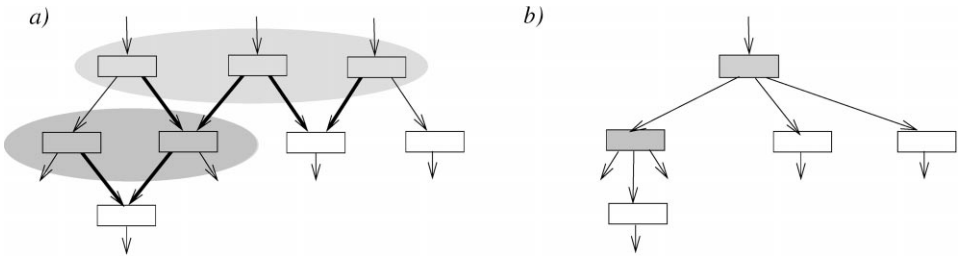

Fig. 14. (a) Equivalent nodes and (b) collapsing equivalent nodes.

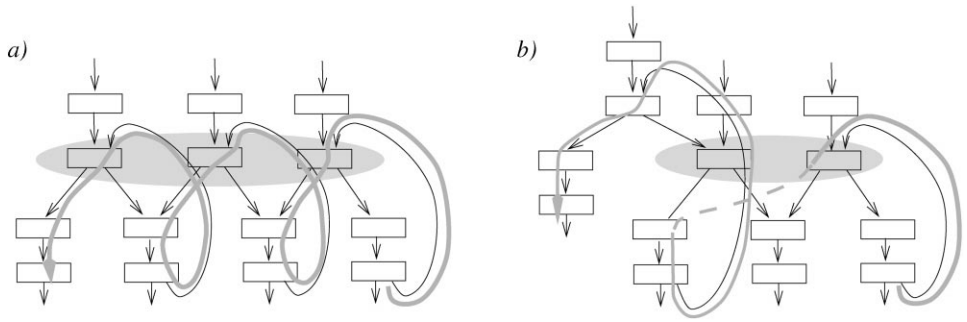

Fig. 15. (a) Decrease of depth due to collapsing of nodes, (b) increase of depth due to collapsing of nodes.

is no longer acyclic, the collapsing process in Part (b) allows to construct a longer acyclic path as indicated by the dashed line connecting two independent acyclic paths.

The inner schedule. The first step of the inner schedule is the easier task. Considering a node $n \in N$ within the collapsed node under consideration and a noncritical edge $(n,m) \in E$ the value of $DnSafe_{\mathtt{Hom}}(n)$ is changed to false if and only if

$$Transp(n) \cdot \overline{DnSafe_{\mathtt{Hom}}(m)}$$

holds. [10]

The second step of the inner schedule is more sophisticated and has to be elaborated with some care. Within any collapsed node side propagation of down-safety information along critical edges is performed by using an exhaustive subiteration process. Information propagation here means that for nodes $n,m \in pred(succ(n))$ in the collapsed node under consideration the value of $DnSafe_{\mathtt{Hom}}(m)$ is changed to false if and only if

$$\overline{Safe_{\mathtt{Hom}}(n)}$$

holds. The crucial point, however, is to organize the information flow along the critical edges. The situation is easy if the zig-zag paths are acyclically shaped as displayed in Figs. 16(a) or (b). In this case the equivalence class can be represented as an undirected tree (see Fig. 17), which can already be built while preprocessing this class. Following the topological order of the tree, information can be propagated completely by a bottom-up traversal (from the leaves to the root) followed by a top-down traversal (from the root to the leaves).

Unfortunately, in general there may be cycles of critical edges. Such situations are illustrated in Figs. 16(c) and (d) where the corresponding (undirected) graphs are depicted in Fig. 17. Hence, in the side-propagation step one is principally faced with a problem of the same difficulty as in the backward propagation of information. However, separating both problems is useful as we expect nested cycles of critical edges to be rare in real-life programs. Nonetheless, it is quite straightforward to cope with this phenomenon. Likewise the acyclic case the equivalence class can be represented as a

[10] Note that there are no noncritical edges directly connecting different nodes of an equivalence class.

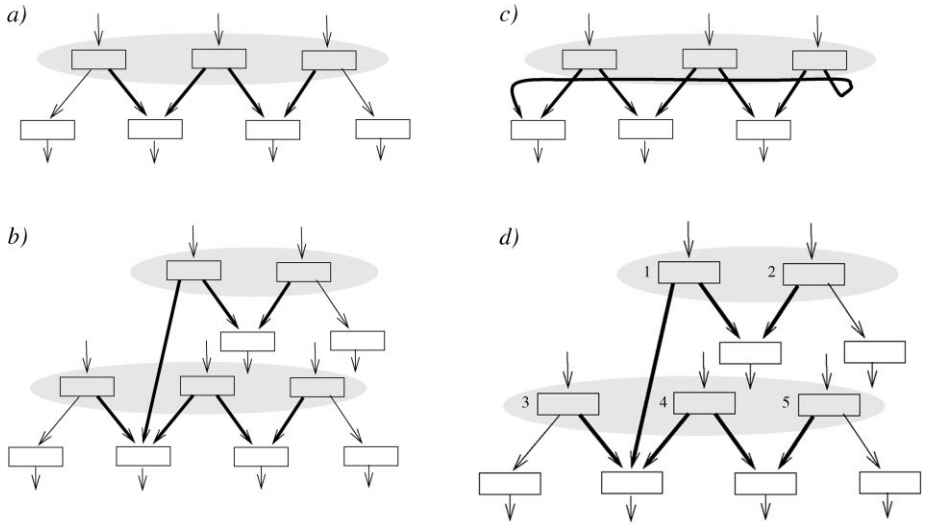

Fig. 16. Shapes of equivalent nodes: (a) chain of critical edges, (b) tree of critical edges, (c) cycle of critical edges and (d) structure with nested cycles of critical edges.

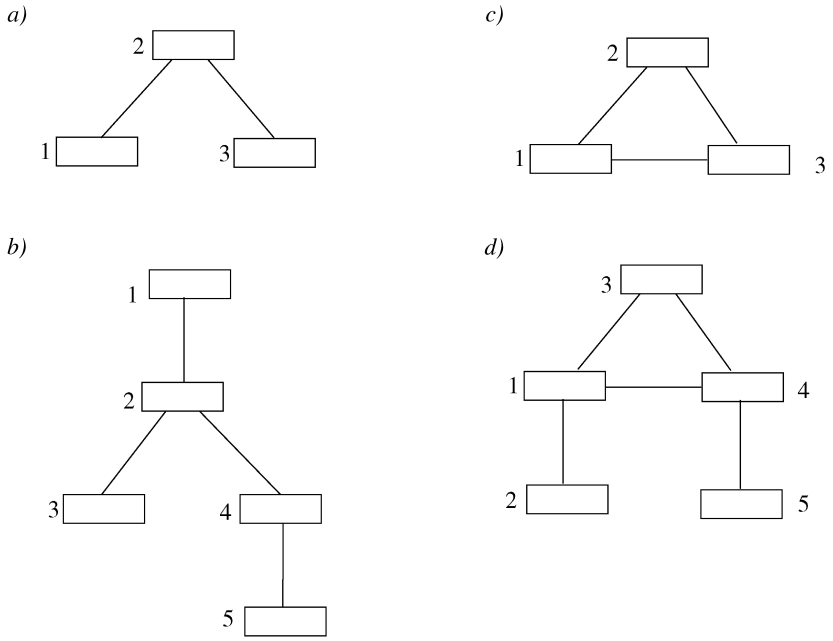

Fig. 17. Equivalent nodes of Fig. 16 in a view as undirected graphs.

tree with some additional nontree edges leading to cycles. The only difference to the noncyclic case is that the tree traversals have to be iterated more than once until the process gets stable. To estimate the number of traversal we borrow the arguments from conventional unidirectional analysis. Denoting the nontree edges within the tree-like representation of an equivalence class as *critical backedges* the number of iterations is bound by d_c, where d_c is the maximum number of critical back edges along an acyclic path in any component representation.

5.4. CLEM

The bidirectional equation system of CLEM in Fig. 10 is also already in a form that makes the side flow component explicit, which makes the techniques presented in Section 5.3.1 also adaptable to this situation. As this process is straightforward its explicit presentation is dispensed.

5.4.1. Complexity of the hybrid iteration strategy

We will discuss the complexity of the hybrid iteration strategy by examining the computation of homogeneous (down-)safety presented in Section 5.3.1. However, dual considerations carry over to the delayability analysis of Section 5.4.

The iteration of homogeneous down-safety in the hybrid approach requires to apply the outer schedule until stabilization. Since the inner schedule propagates the information completely within each collapsed node the overall effort can be estimated by

$$(d' + 2)(e_{nc} + 2(d_c + 2)e_c)$$

bit-vector steps, where e_{nc} and e_c denote the number of noncritical and critical edges, respectively, d' is the depth of the collapsed flow graph G' and d_c the critical depth as defined in Section 5.3.1.

It is commonly argued that the depth of a flow graph is a reasonably small constant in practice. We already discussed that d_c is at least as likely to be a small constant, too. Hence the algorithm is expected to behave linear in e for real-life programs. In particular, we succeed in giving the first linear worst-case estimation for acyclic programs like the one of our introductory example in Fig. 2.

6. Conclusion

We presented an adaption of lazy expression motion to flow graphs with critical edges (CLEM) as a model how to cope with bidirectional dependencies in expression motion. On the conceptual level we isolated homogeneity requirements as the source for bidirectional dependencies. Based upon a clean specification we presented three alternative approaches for computing CLEM: a bidirectional, a unidirectional and a hybrid one. In particular, the latter one led to a new hybrid iteration strategy which is almost as fast as its unidirectional counterparts in the absence of critical edges. This

dramatically improves all known estimations for bidirectional bit-vector methods. Although this may sound like an advocacy for the non-splitting of critical edges it should kept in mind that the computational quality of expression motion in the absence of critical edges is unexcelled when compared to the situation where critical edges are present. Moreover, other problems not discussed in the paper give strong evidence for the vantages of splitting of critical edges. For instance, the extension of lazy expression motion to composite expression in [26] only succeeds in minimizing the overall amount of temporaries if critical edges are absent. However, it is our hope that any implementation of code motion techniques where one is obliged to cope with critical edges may benefit from the ideas presented in this paper.

Appendix

Proof of Theorem 8. The proof can be structured into three subparts:
(1) CBEM is admissible,
(2) CBEM is profitable,
(3) CBEM is computationally optimal.

(1) *Admissibility*: We have to prove:
(a) $Insert_{\mathrm{CBEM}}(n) \Rightarrow Safe(n)$,
(b) $Replace_{\mathrm{CBEM}}(n) \Rightarrow Correct_{\mathrm{CBEM}}(n)$.
Part (a) is a simple consequence of the following trivial sequence of implications:

$$Insert_{\mathrm{CBEM}}(n) \;\Rightarrow\; Earliest_{\mathtt{Hom}}(n) \;\Rightarrow\; Safe_{\mathtt{Hom}}(n) \;\Rightarrow\; Safe(n).$$

Instead of proving Part (b) directly the implication is generalized towards:

$$Safe_{\mathtt{Hom}}(n) \;\Rightarrow\; Correct_{\mathrm{CBEM}}(n). \tag{A.1}$$

By the definition of correctness this can be rewritten as

$$\forall p \in \boldsymbol{P}[s,n].\ Safe_{\mathtt{Hom}}(n) \Rightarrow \exists i \leqslant \ell_p.\ Earliest_{\mathtt{Hom}}(p_i) \wedge \forall i < j \leqslant \ell_p.\ \neg Earliest_{\mathtt{Hom}}(p_j).$$

This implication can then be proved by induction on ℓ_p which is straightforward by using the following property that directly follows from the definitions of $DnSafe_{\mathtt{Hom}}$, $UpSafe_{\mathtt{Hom}}$ and $Earliest_{\mathtt{Hom}}$, respectively.

$$Safe_{\mathtt{Hom}}(n) \;\Rightarrow\; (Earliest_{\mathtt{Hom}}(n) \;\vee\; \forall m \in pred(n).\ Safe_{\mathtt{Hom}}(m))$$

(2) *Profitability*: First we show

$$Earliest_{\mathtt{Hom}}(n) \;\Rightarrow\; DnSafe_{\mathtt{Hom}}(n). \tag{A.2}$$

Suppose $Earliest_{\mathtt{Hom}}(n)$ and $UpSafe(n)$ holds. The definition of $Earliest_{\mathtt{Hom}}$ implies $\neg UpSafe(m)$ for any predecessor m of n. On the other hand, $UpSafe(n)$ then

forces that $Comp(m)$ and $Transp(m)$ holds for any $m \in pred(n)$. Due to the node splitting preprocess (cf. Section 2.1) there can be only one such m and this m does not have other successors than n. Hence by the definition of $DnSafe_{\mathtt{Hom}}$ this establishes

$$DnSafe_{\mathtt{Hom}}(m) \wedge Transp(m),$$

which would mean that $Earliest_{\mathtt{Hom}}(n)$ cannot hold in contradiction to our assumption. Hence the assumption on $UpSafe(n)$ was wrong, and thus $DnSafe_{\mathtt{Hom}}(n)$ must hold.

Based on Implication (A.2) we prove the more general proposition

$$DnSafe_{\mathtt{Hom}}(n) \;\Rightarrow\; Profitable_{\mathrm{CBEM}}(n), \tag{A.3}$$

instead of

$$Insert_{\mathrm{CBEM}}(n) \;\Rightarrow\; Profitable_{\mathrm{CBEM}}(n).$$

Using the definition of $Profitable_{\mathrm{CBEM}}$ this can be rewritten as

$$\begin{aligned}&\forall p \in \boldsymbol{P}[n, \boldsymbol{e}].DnSafe_{\mathtt{Hom}}(n)\\ &\quad\Rightarrow\; (\exists i \leqslant \ell_p.\; Comp(p_i) \;\wedge\; \forall 1 < j \leqslant i.\; \neg Earliest_{\mathtt{Hom}}(p_j))\end{aligned}$$

and proved by an induction on ℓ_p. This, however, easily follows from the property below which is straightforward from the definition of $DnSafe_{\mathtt{Hom}}$.

$$\begin{aligned}DnSafe_{\mathtt{Hom}}(n) \Rightarrow (&Comp(n) \;\vee\; Transp(n) \;\wedge\\ &\forall m \in succ(n).\; DnSafe_{\mathtt{Hom}}(m)).\end{aligned}$$

(3) *Computational optimality*: As in its "noncritical" counterpart (cf. [18]) the argument of computational optimality is based on the observation that any computationally optimal expression motion transformation must have a computation between an earliest computation point and its corresponding first-use sites. Formally, this is captured by the notion of *homogeneous earliest first-use paths* ($\mathrm{EFU}_{\mathtt{HOM}}$ paths). A path p is an $\mathrm{EFU}_{\mathtt{HOM}}$ path iff

$$Earliest_{\mathtt{Hom}}(p_1) \;\wedge\; Comp(p_{\ell_p}) \;\wedge\; \forall 1 < i \leqslant \ell_p.\; \neg Earliest_{\mathtt{Hom}}(p_i).$$

In addition, we use a property which is a direct consequence of the definition of $DnSafe_{\mathtt{Hom}}$:

$$\neg DnSafe_{\mathtt{Hom}}(n) \;\Rightarrow\; \exists k \in \mathbb{N},\; m \in (pred \circ succ)^k(n).\; \neg Safe(m).$$

For a node n with $\neg DnSafe_{\mathtt{Hom}}(n)$ let us denote the smallest such k by k_n. Then we are going to prove by induction on k_n:

$$\neg Safe_{\mathtt{Hom}}(n) \;\Rightarrow\; \neg Correct_{\mathrm{EM}}(n). \tag{A.4}$$

If $k_n = 0$ this reduces to the premise $\neg Safe(n)$ which, exactly as in the noncritical setting, immediately implies $\neg Correct_{\mathrm{EM}}(n)$ (cf. Theorem 3.5 in [18]). On the other hand, if $k_n > 0$, there is a node $m \in pred(succ(n))$ with $\neg Safe_{\mathtt{Hom}}(m)$ and $k_m < k_n$ which, by the induction hypothesis, concludes the proof of (A.4).
With Implication (A.4) it is finally easy to see that any computational optimal expression motion transformation must have a computation (i.e. either an insertion or a nonremoved original computation) on an $\mathrm{EFU}_{\mathtt{HOM}}$ path: obviously, a predecessor m of p_1 violates EM-correctness, i.e. $\neg Correct_{\mathrm{EM}}(m)$ holds, which directly forces that an EM-computation must be situated on p. □

Proof of Theorem 10. The proof can be structured into four subparts:
(1) CLEM is admissible.
(2) CLEM is profitable.
(3) CLEM is computationally optimal.
(4) CLEM is lifetime optimal.

(1) *Admissibility*: We have to prove:
(a) $Insert_{\mathrm{CLEM}}(n) \Rightarrow Safe(n)$,
(b) $Replace_{\mathrm{CLEM}}(n) \Rightarrow Correct_{\mathrm{CLEM}}(n)$.
Part (a) is a consequence of the following sequence of implications:

$$\begin{aligned} Insert_{\mathrm{CLEM}}(n) &\Rightarrow Latest_{\mathtt{Hom}}(n) \;\Rightarrow\; Delayed_{\mathtt{Hom}}(n) \\ &\Rightarrow DnSafe_{\mathtt{Hom}}(n) \;\Rightarrow\; DnSafe(n) \;\Rightarrow\; Safe(n). \end{aligned}$$

Apart from the implication

$$Delayed_{\mathtt{Hom}}(n) \;\Rightarrow\; DnSafe_{\mathtt{Hom}}(n), \tag{A.5}$$

which shall be proved in the following all other implications are trivial. Obviously, $Delayed_{\mathtt{Hom}}(n)$ implies that there is a path in $\boldsymbol{P}[\boldsymbol{s}, n]$ such that

$$\exists i \leqslant \ell_p.\ Earliest_{\mathtt{Hom}}(p_i) \wedge \forall\, i \leqslant j < \ell_p.\ \neg Comp(p_j).$$

Then we may show by an induction on j

$$\forall\, i \leqslant j \leqslant \ell_p.\ DnSafe_{\mathtt{Hom}}(p_j),$$

which is almost trivial exploiting Implication (A.2) and the obvious implication

$$DnSafe_{\mathtt{Hom}}(n) \;\Rightarrow\; (Comp(n) \;\vee\; \forall m \in succ(n).\ DnSafe_{\mathtt{Hom}}(m)\,).$$

With $Comp(n) \Rightarrow DnSafe_{\mathtt{Hom}}(n)$ Part (b) follows directly from Implication (A.1) which is included in the proof of Theorem 8 (cf. Section 6).

(2) *Profitability*: Here we have the following sequence of implications:

$$Latest_{\mathtt{Hom}}(n) \;\Rightarrow\; Delayed_{\mathtt{Hom}}(n) \;\Rightarrow\; DnSafe_{\mathtt{Hom}}(n) \;\Rightarrow\; Profitable_{\mathrm{CLEM}}(n),$$

where the second and third one are already proved as Implication (A.5) and (A.3), respectively.

(3) *Computational optimality*: For proving computational optimality it is enough to show that every $\mathrm{EFU}_{\mathtt{HOM}}$ path p (see p. 30) contains exactly one program point that satisfies $Latest_{\mathtt{Hom}}$. It is trivial that there is at least one such point. Let i be the smallest index with $Latest_{\mathtt{Hom}}(p_i)$. If $i < \ell_p$ the homogeneity of $Delayed_{\mathtt{Hom}}$ implies that $\neg Delayed_{\mathtt{Hom}}(p_{i+1})$ holds. An easy induction proof then yields $\neg Latest_{\mathtt{Hom}}(p_j)$ for all $i < j \leqslant \ell_p$.

(4) *Lifetime optimality*: Like in its "noncritical" counterpart (cf. [18]) it is essentially enough to show that any computational optimal expression motion EM must place inside of the range of delayable program points,[11] i.e.

$$Insert_{\mathrm{EM}}(n) \;\Rightarrow\; Delayed_{\mathtt{Hom}}(n). \tag{A.6}$$

Therefore, we use the following property which is a direct consequence of the definition of $Delayed_{\mathtt{Hom}}$:

$$\neg Delayed_{\mathtt{Hom}}(n) \;\Rightarrow\; \exists k \in \mathbb{N},\ m \in (succ \circ pred)^k(n).\ \neg Delayed(m).$$

For a node n with $\neg Delayed_{\mathtt{Hom}}(n)$ let us denote the smallest such k by k_n. Then we are going to prove by the contrapositive of Implication (10) by an induction on k_n.

If $k_n = 0$ this reduces to the premise $\neg Delayed(n)$ which, exactly as in the noncritical setting, immediately implies $\neg Insert_{\mathrm{EM}}(n)$ (cf. Theorem 3.16(3) in [18]). On the other hand, if $k_n > 0$, there is a node $m \in succ(pred(n))$ with $\neg Delayed_{\mathtt{Hom}}(m)$ and $k_m < k_n$ which, by the induction hypothesis, concludes our proof. □

References

[1] F. Chow, A portable machine independent optimizer – Design and measurements, Ph.D. thesis, Standford University, Dept. of Electrical Engineering, Stanford, CA, 1983. Published as Tech. Rep. 83–254, Computer Systems Lab., Stanford University.

[2] D.M. Dhamdhere, A fast algorithm for code movement optimization, ACM SIGPLAN Notices 23 (10) (1988) 172–180.

[3] D.M. Dhamdhere, A new algorithm for composite hoisting and strength reduction optimisation (+ Corrigendum), Int. J. Comput. Math. 27 (1989) 1–14(+31 − 32).

[4] D.M. Dhamdhere, A usually linear algorithm for register assignment using edge placement of load and store instructions, J. Comput. Languages 15 (2) (1990) 83–94.

[5] D.M. Dhamdhere, Practical adaptation of the global optimization algorithm of Morel and Renvoise, ACM Trans. Programm. Languages Systems 13 (2) (1991) 291–294. Technical Correspondence.

[6] D.M. Dhamdhere, U.P. Khedker, Complexity of bidirectional data flow analysis, Conf. Record of the 20th ACM Symp. on the Principles of Programming Languages (POPL), Charleston, SC, January 1993, pp. 397–409.

[7] D.M. Dhamdhere, H. Patil, An elimination algorithm for bidirectional data flow problems using edge placement, ACM Trans. Programm. Languages Systems 15 (2) (1993) 312–336.

[11] The reasoning on the suppression of initializations at isolated program points is not influenced by the presence of critical edges. We therefore refer to [18] on this point.

[8] D.M. Dhamdhere, B.K. Rosen, F.K. Zadeck, How to analyze large programs efficiently and informatively, Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)'92, ACM SIGPLAN Notices, Vol. 27,7, San Francisco, CA, June 1992, pp. 212–223.
[9] K.-H. Drechsler, M.P. Stadel, A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies", ACM Trans. Programm. Languages Systems 10 (4) (1988) 635–640. Technical Correspondence.
[10] K.-H. Drechsler, M.P. Stadel, A variation of Knoop, Rüthing and Steffen's lazy code motion, ACM SIGPLAN Notices 28 (5) (1993) 29–38.
[11] M.S. Hecht, J.D. Ullman, A simple algorithm for global data flow analysis problems, SIAM J. Comput. 4 (4) (1977) 519–532.
[12] S. Horwitz, A. Demers, T. Teitelbaum, An efficient general iterative algorithm for data flow analysis, Acta Inform. 24 (1987) 679–694.
[13] S.M. Joshi, D.M. Dhamdhere, A composite hoisting-strength reduction transformation for global program optimization – Part I, Int. J. Comput. Math. 11 (1982) 21–41.
[14] S.M. Joshi, D.M. Dhamdhere, A composite hoisting-strength reduction transformation for global program optimization – Part II, Int. J. Comput. Math. 11 (1982) 111–126.
[15] U.P. Khedker, D.M. Dhamdhere, A generalized theory of bit vector data flow analysis, ACM Trans. Programm. Languges Systems 16 (5) (1994) 1472–1511.
[16] J. Knoop, O. Rüthing, B. Steffen, Lazy code motion, Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)'92, ACM SIGPLAN Notices, Vol. 27,7, San Francisco, CA, June 1992, pp. 224–234.
[17] J. Knoop, O. Rüthing, B. Steffen, Lazy strength reduction, J. Programm. Languages 1 (1) (1993) 71–91.
[18] J. Knoop, O. Rüthing, B. Steffen, Optimal code motion: theory and practice, ACM Trans. Programm. Languages Systems 16 (4) (1994) 1117–1155.
[19] J. Knoop, O. Rüthing, B. Steffen, Partial dead code elimination, Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)'94, ACM SIGPLAN Notices, Vol. 29,6, Orlando, FL, June 1994, pp. 147–158.
[20] J. Knoop, O. Rüthing, B. Steffen, The power of assignment motion, Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)'95, ACM SIGPLAN Notices, Vol. 30,6, La Jolla, CA, June 1995, pp. 233–245.
[21] J. Knoop, O. Rüthing, B. Steffen, Code motion and code placement: just synonyms? Proc. 6th European Symp. on Programming (ESOP), Lecture Notes in Computer Science, Vol. 1381, Lisbon, Portugal, Springer, Berlin, 1998, pp. 154–196.
[22] P.M. Masticola, T.J. Marlowe, B.G. Ryder, Lattice frameworks for multisource and bidirectional data flow problems, ACM Trans. Programm. Languages Systems 17 (5) (1995) 777–802.
[23] E. Morel, Data flow analysis and global optimization, in: B. Lorho (Ed.), Methods and Tools for Compiler Construction, Cambridge University Press, Cambridge, 1984.
[24] E. Morel, C. Renvoise, Global optimization by suppression of partial redundancies, Comm. ACM 22 (2) (1979) 96–103.
[25] E. Morel, C. Renvoise, Interprocedural elimination of partial redundancies, in: S.S. Muchnick, N.D. Jones (Eds.), Program Flow Analysis: Theory and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1981, pp. 160–188 (Chapter 6).
[26] O. Rüthing, Interacting code motion transformations: their impact and their complexity, Ph.D. thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Kiel, Germany, 1997. Lecture Notes in Computer Science, Vol. 1539, Springer, Heidelberg, 1998.
[27] O. Rüthing, Bidirectional data flow analysis in code motion: myth and reality, Proc. 5th Int. Static Analysis Symp. (SAS'98), Lecture Notes in Computer Science, Vol. 1503, Pisa, Italy, Springer, Berlin, September 1998, pp. 1–16.
[28] O. Rüthing, Optimal code motion in the presence of large expressions, Proc. IEEE Int. Conf. on Computer Languages, Chicago, Il., May 1998, pp. 216–225.
[29] O. Rüthing, J. Knoop, B. Steffen, Sparse code motion, Conf. Record of the 27th ACM Symp. on the Principles of Programming Languages (POPL), Boston, MA, January 2000, to appear.
[30] A. Sorkin, Some comments on a solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies", ACM Trans. Programm. Languages Systems 11 (4) (1989) 666–668. Technical Correspondence.